# 16

# Enterprise Blockchain

In this chapter, we'll investigate **enterprise blockchains**. What is the standard architecture of enterprise blockchains? Why are they required? We will answer these questions and will also try to answer the big question of why current public blockchains are not necessarily a suitable choice for enterprise use cases.

We will also introduce several enterprise blockchain platforms, including Quorum. Along the way, we'll cover the following topics:

- Enterprise solutions and blockchain
- Limiting factors
- Requirements
- Enterprise blockchain versus public blockchain
- Use cases of enterprise blockchains
- Enterprise blockchain architecture
- Designing enterprise blockchain solutions
- Blockchain in the cloud
- Currently available enterprise blockchains
- Enterprise blockchain challenges
- Quorum
- VMware Blockchain
- Setting up Quorum with IBFT
- Other Quorum projects

We discussed several different blockchain types in *Chapter 1*, *Blockchain 101*, including permissioned, public, private, and consortium blockchains. Enterprise blockchains are a permissioned **consortium** chain type that primarily tackles enterprise requirements.

Permissioned doesn't mean private. Permissioned blockchains can also be public and allow access only to known participants. Enterprise blockchains are usually private and permissioned and are run between consortium members.

While public blockchains provide integrity, consistency, immutability, and security guarantees, they lack certain features, which makes them less suitable for enterprise usage. We'll discuss these limitations in detail shortly.

First, we'll look into what enterprise solutions are and how blockchain can fit into an enterprise. Second, we'll see what questions should be answered before introducing an enterprise blockchain solution to a business.

# Enterprise solutions and blockchain

**Enterprise solutions** integrate different fragments of a business and enable it to achieve its goals by providing business-critical information to the stakeholders. Here, we'll consider the question: *Does blockchain fit this definition and help achieve enterprise business goals?*

To find out if a blockchain is suitable for an enterprise or not, we can ask some questions:

- How can business/enterprise processes be improved using blockchain technology?
- Do I really need an enterprise blockchain? For this, we can ask some questions to rationalize whether we need an enterprise blockchain solution or not. Does my use case:
    - Need shared data between participants?
    - Have participants from other organizations that are not necessarily trustworthy and have conflicting interests?
    - Need strict auditing? A blockchain can provide this, as it is an immutable and tamper-proof chain of records that can provide a definite audit trail of activities performed on the chain (enterprise system).
    - Need to ensure the confidentiality of transactions?
    - Need to ensure the anonymity of participants?
    - Need controlled but transparent updates to the ledger?
    - Not need a single trusted authority? Instead, the decisions on the network (updates to the ledger) should be consortium member-driven and agreed upon between members.

If the answer to any of the preceding questions is *yes*, then using an enterprise blockchain solution could be a good option. Otherwise, a traditional database might give a better alternative. In addition to the questions mentioned here, when proposing an enterprise blockchain solution, we also need to answer some other questions from a business perspective that help establish the overall vision and strategy of the implementation of the blockchain solution:

- What is the overall objective of the proposed blockchain solution? Does it align with the business goals of the enterprise? Is it a **Proof of Concept** (**PoC**) project, intended just to demonstrate an idea, or a production project with real business deliverables?
- Where would it be deployed? Cloud or local hosting? Who will manage the blockchain solution once it is deployed?

- Risk management considerations—does the solution follow any established guidelines for risk management? For example, NIST 800-37 (`https://csrc.nist.gov/publications/detail/sp/800-37/rev-2/final`) is a risk management framework that provides a process used to manage security and privacy risk for IT systems and organizations.
- Are there any other projects that this organization may have implemented already using enterprise blockchain? Can we learn from previous experiences and leverage some of the resources and best practices that may have been developed previously?

These questions should be—and are usually—asked when developing any other enterprise solution, but in relation to blockchain, these questions become even more critical due to the nascent and immature nature of enterprise blockchain technology. A clear definition of the objectives, along with a clear alignment with business requirements, will result in an implementation that meets business goals.

Next, we'll envisage some success factors that can help enterprise blockchains to become successful and adopted in the enterprise.

## Success factors

There are some business-oriented factors that should be addressed for a successful enterprise blockchain solution.

Of the utmost importance is the requirement that the blockchain solution should bring some economic value and help to achieve real business goals. Moreover, enterprise solutions should be aligned with business goals.

The primary value of enterprise blockchains is in the property of being a sharable, replicable, permissioned ledger between organizations that immediately results in cost reduction by eliminating the need for data exchange. In doing so, we also eliminate the need for infrastructure and tools to support such exchanges. Also, due to the security, immutability, and auditing provided as inherent features of a blockchain, there is no need to invest separately for these requirements.

Blockchain solutions that integrate easily/seamlessly with existing systems provide better value because already mature legacy systems (at least at this stage) provide a mechanism to connect with the blockchain, read its data, and save it in a known format that the enterprise is already familiar with. Moreover, just a blockchain network on its own is not entirely useful in enterprises if it is not integrated with existing back-office systems such as **Enterprise Resource Planning** (**ERP**), backend databases, record reconciliation systems, or other organization reporting tools.

An enterprise blockchain solution should be seen as a complete end-to-end enterprise solution as part of the larger enterprise architecture, instead of only a siloed blockchain network. We'll explore this topic later in this chapter, under the *Designing enterprise blockchain solutions* section.

Enterprise-grade governance, control, and security are also desirable features from a business perspective, as they allow business stakeholders to apply already-established organization rules and policies to the blockchain. This can also help to achieve regulatory and compliance requirements.

Now, the following question arises: *With the availability of many different public blockchain platforms, why has the adoption of blockchain technology in enterprises still not been fully achieved?* The reason is that there are several factors that make public blockchains unsuitable for enterprise use cases. In the next section, we'll answer this question and explore why public blockchains are not quite suitable for enterprise use cases.

## Limiting factors

We discussed several benefits of blockchain technology in general in *Chapter 1*, *Blockchain 101*. While all those benefits are attainable using public blockchains, several features are lacking in public blockchains, which makes them unsuitable for use in enterprise use cases. The interest in enterprise blockchain arises from these limitations in public blockchains, along with specific requirements in any business.

We'll describe some of the most common concerns next:

- Slow performance: Public blockchains are slow and can process only a few transactions per second. Bitcoin processes 3-4 transactions per second, while Ethereum processes around 14. This low transaction rate is not suitable for businesses that usually require high transaction speed. For example, card payment businesses typically need to process thousands of transactions per second.
- Lack of access governance: Public blockchains are available for anyone to join, which makes it easier for investors and cryptocurrency enthusiasts to join and helps with the network effect. However, in an enterprise, all participants must be known so that everyone knows who they are dealing with. This lack of an identification and access control mechanism makes public blockchains rather unsuitable for businesses.
- Lack of privacy: Public blockchains are inherently transparent, and everything on the ledger is visible to everyone. This means anyone can easily view transaction details and participants involved in a transaction. Lack of privacy results in a strategy leak problem, which we describe next.
- Strategy leak problem: As public blockchains are open to everyone, it is possible that by using traffic analysis, one can figure out which party is communicating with which other party, and thus can infer who is doing more business. Moreover, even a simple transaction count from an address can be used to infer the scale of the business, and thus the revenue an entity might be generating.
- Probabilistic consensus: Traditional public blockchains usually use a **Proof of Work** (**PoW**) type of consensus mechanism, which is inherently a probabilistic protocol that provides probabilistic finality. Even though the confirmations mechanism, as discussed in *Chapter 6*, *Bitcoin Architecture*, does give a certain level of confidence that a transaction is irrevocable, it is still possible that the chain may fork and transactions can be lost. This issue is particularly a concern for businesses where once a transaction commits, it is deemed final. Imagine receiving a property ownership document on a chain from someone only to discover later that the blockchain has forked, and that you are no longer the owner of the property.
- To address this limitation, enterprise blockchains use deterministic consensus algorithms, which provide immediate finality.

- Note that some newer public blockchains such as Avalanche and Cosmos do have deterministic finality. However traditionally public chains have been and some still are PoW, e.g., Bitcoin.
- Transaction fees: In Ethereum or other similar blockchains, the transaction fee is charged in the native cryptocurrency for each transaction execution. While this mechanism provides incentives for miners and protects against spam, there is no requirement for such an arrangement in enterprise blockchains. If an organization were to use public blockchains for their business transactions, then they'd need cryptocurrency in reserve to pay for operations on the blockchain. This extra cost might be undesirable for some businesses. Moreover, when the network is busy, the gas fees go up significantly, which results in even more costs.
- Network congestion: It is possible on public blockchain networks that, due to busy dApps, my own dApp is impacted because the network is busy. If the whole network is congested due to too many transactions, then all dApps would run slow, and this is the risk that can prove detrimental to a business. While the concerns mentioned here are considered limitations in public blockchains, based on these concerns and limitations, we can derive and define several requirements, or features, of a blockchain that will enable it to become an enterprise blockchain. In other words, the limitations in public blockchains can be seen as requirements of enterprise blockchains.

The next section talks about several features that a blockchain should possess to become suitable for enterprises.

## Requirements

In addition to *integrity* and *consistency*, which are also provided by public blockchains, there are several other requirements specifically for enterprise blockchains that make them suitable for enterprise use cases. In some cases, some requirements become even stricter in enterprise blockchains compared to public blockchains. For example, in public blockchains, eventual consistency is acceptable. However, in enterprise blockchains, the moment a transaction is committed, it should immediately finalize and irrevocably become part of the global record (state). Thus, we'll begin by briefly defining integrity and consistency before introducing specific requirements for enterprise blockchains:

- **Integrity**: This attribute of a blockchain is provided by the use of hash functions and digital signatures, and plays a vital role in the overall security of the blockchain. Hash functions allow us to check for any data modifications, whereas digital signatures ensure that the messages that originated from a sender have not been altered.
- **Consistency**: This attribute of a blockchain ensures that all honest nodes agree on the same sequence of blocks. To achieve this, various mechanisms are used, such as PoW or **Byzantine Fault Tolerance** (**BFT**) consensus protocols.

You can refer to *Chapter 1*, *Blockchain 101*, for a refresher on this, as we introduced these properties there in more detail.

Now, we'll introduce three fundamental requirements that should be met for a blockchain to become suitable for enterprises. These requirements are **privacy**, **performance**, and **access governance**.

## Privacy

Privacy is of paramount importance in enterprise blockchains. Privacy has two facets: first, *confidentiality*, and second, *anonymity*.

**Confidentiality** is a fundamental requirement in an enterprise. It is anticipated that, in enterprise blockchains, all transactions hide their payloads so that the value of the transactions is not revealed to anyone who is not privy to the transaction.

Private transactions can be defined as transactions that meet the privacy requirements of an enterprise use case. There are two types of private transactions, as defined by **Enterprise Ethereum Alliance** (**EEA**):

- **Restricted private transactions**: This type of private transaction is transmitted only to those parties on the blockchain network who are privy to the transaction.
- **Unrestricted private transactions**: Under the unrestricted private transaction paradigm, private transactions are transmitted to all participants on the network, regardless of whether they are privy to the transaction or not. The payload is still encrypted and confidential, but the transaction itself is broadcast to the entire network.

There are many approaches to achieving privacy in enterprise blockchains and blockchain in general. These methods range from an off-chain mechanism like privacy managers (used in Quorum and some other enterprise chains) to utilizing **zero-knowledge proofs** (**ZKPs**), trusted hardware, and **secure multiparty computation** (**SMPC**). We'll cover some of these techniques in the next chapter, *Chapter 17*, *Scalability*, but in this chapter, we will mainly focus on privacy manager-based privacy, where an off-chain component is used to provide privacy services. We will discuss the privacy manager-based approach in the *Quorum* section later in this chapter.

**Anonymity** in enterprise blockchains might not seem a strict requirement at first, because all participants are known and identified. However, it is necessary for scenarios with some competing participants and conducting business on-chain. It could be essential, for example, that in a scenario where parties *X* and *Y* are transacting together, party *Z* doesn't find out which parties are transacting together. Even though the transaction values are not visible, they can still reveal details about the business that the two parties might be doing. If that information is available publicly, then other parties on the consortium chain will gain market intelligence and may try to influence that process via marketing or other methods.

## Performance

Due to the high-speed requirements of businesses, enterprise blockchains must be able to process transactions at a high rate.

Performance has two facets: scalability and speed. **Speed** deals with how many transactions can be processed in a given amount of time. It deals with the ability of a system to handle a large volume of transactions at an acceptable speed.

On the other hand, we have the number of participants in a system. Public blockchains can support a large number of users. This is especially true in cryptocurrency blockchains such as Bitcoin and Ethereum.

This is possible due to the PoW or **Proof of Stake** (**PoS**) algorithms for consensus used in these public blockchains. However, in enterprise blockchains, a different class of consensus algorithms (most commonly, BFT) are used, which do not work well with a large number of nodes. This results in limiting the number of users on a consortium chain.

Here, we have to consider a tradeoff. Ideally, an enterprise blockchain should be able to perform well with a large number of users; however, with the tradeoff, enterprise blockchains should be able to process transactions at a high rate. This is what most enterprise blockchains focus on since, in many use cases, the number of nodes is not that high, and speed can be prioritized over scalability.

In public blockchains, sometimes, network congestion caused by high volumes of traffic can cause performance issues and can increase transaction processing times. This is detrimental to **enterprise dApps**, which need faster transaction processing and response times. Network congestion also results in increased gas prices, which can also be a concern for businesses from a cost perspective.

## Access governance

From another angle, being a permissioned blockchain, enterprise-grade access control (in the form of either a new mechanism on the chain or control driven by an enterprise SSO already in place) is a fundamental requirement in enterprise blockchains. As all participants must be identifiable on a consortium chain, it is essential to build an access control mechanism that facilitates that process. This feature can be achieved by using enterprise-grade access control mechanisms such as **Role-Based Access Control** (**RBAC**).

RBAC is an ANSI standard. More information is available in the ANSI INCITS 359-2004 document. You can find more information on the document and the RBAC standard here: `https://csrc.nist.gov/projects/role-based-access-control`.

The access control mechanism also can address **Know Your Customer** (**KYC**) requirements.

In addition to privacy, performance, and access governance, which are usually identified as three fundamental requirements of enterprise blockchains, there are also some other requirements that are highly desirable but can be considered somewhat optional, as they are more use case dependent. We'll describe these next.

## Further requirements

In this section, we'll present some further requirements that are very useful and can increase the suitability/efficacy of enterprise blockchain solutions.

## Compliance

A common concern is compliance. Public blockchains are not suitable for enterprise use cases due to strict regulatory and law requirements in almost all sectors such as finance, health, and government. Compliance challenges mainly include regulatory compliance, standards compliance, data sovereignty, and liability concerns.

We'll define these briefly next:

- **Compliance with standards and regulations**: Often, in enterprise use cases, compliance with a technical standard or laws is required. For example, compliance with GDPR is mandatory in the European Union and the European Economic Area. Another example is compliance with **Financial Conduct Authority** (**FCA**) regulations in the UK. Moreover, it might be necessary, in certain use cases, to comply with technical standards such as cryptography standards published by NIST. One such example is the use of NIST-approved curves, as described here: `https://csrc.nist.gov/Projects/elliptic-curve-cryptography`.
- **Data sovereignty**: Data sovereignty is a broad topic that mainly subjects data to the laws of the country in which it is located. For example, under GDPR, transferring personal data outside the EU is subject to adequacy decisions (Article 45) and the appropriate safeguards (Article 46).

More on this GDPR guidance can be found here: `https://gdpr.eu/tag/chapter-5/`.

  As public blockchains are borderless and geographically dispersed systems, compliance with such regulations can be challenging.

- **Liability**: In traditional business and IT systems, legal responsibility often lies with a party who provides a specific service; for example, a cloud service provider is responsible for handling data in accordance with local laws and regulations. In a decentralized public blockchain, the data is on a public blockchain, and it, therefore, becomes challenging to keep a single party responsible for data management or providing services. In case of malicious incidents, again, it is not possible to hold any single party responsible.

This limitation poses a significant challenge in enterprise settings where, usually, a responsible party is in control of service provision and is held accountable. From another angle, we know that, in traditional systems, in case of any problems, a legal system can help. However, in a decentralized blockchain, it can become quite challenging to blame any single party for their actions. Therefore, in enterprise blockchains, the ability to comply with regulations and laws becomes a very sought-after attribute and is achievable by appropriate identification, KYC, and access control mechanisms. In an enterprise chain, the participants are known and can be held accountable for their actions.

## Interoperability

As the enterprise blockchain ecosystem evolves, the need to be able to exchange data between disparate enterprise and public blockchains also arises. Lack of standardization also alleviates this problem; however, standardization efforts are underway, such as EEA (discussed in the *Enterprise blockchain challenges* section). There are also interoperability solutions being developed and available for blockchains that allow interoperability between chains, such as **ION**, **IBC protocol**, **Polkadot**, and **Interledger**.

## Integration

No enterprise blockchain is an isolated end-to-end solution. It has to integrate with existing enterprise systems or other off-chain systems that are part of the whole enterprise solution to fulfill business goals. Therefore, enterprise blockchains must provide interfaces for integration. This can be as simple as providing RPC endpoints, or as complex as building blockchain-specific connectors and plugins to integrate with enterprise service buses or other legacy systems—more on this in the *Enterprise blockchain architecture* section.

Integration with security devices such as **Hardware Security Modules** (**HSMs**) is also quite desirable for many enterprise use cases where strict security is required, or due to regulatory or compliance requirements.

## Ease of use

Usually, enterprise systems are easy to deploy and use. Deployment in enterprises is easy and quick and often relies on mature frameworks and tools, such as Ansible and other proprietary tools, but this is not the case with blockchain.

The deployment of enterprise systems is a well-studied, understood, and mature area. With enterprise orchestration tools and established techniques over the years, enterprise deployment has become easy. However, blockchains are not as easy to deploy as other enterprise systems. With the availability of **Blockchain as a Service** (**BaaS**) and deployment automation tools, this is changing. However, there is still some work that needs to be done.

## Monitoring

Monitoring using visualization tools plays a vital role in any enterprise solution. Without the ability to monitor and visualize a system, it is almost impossible to ensure the health of the enterprise system. It is also a desirable feature in enterprise blockchain solutions to be able to visualize the blockchain network. Monitoring a blockchain allows an administrator to keep an eye on the network's health and operations. It allows an administrator to monitor and respond to the events of interest, such as a node going down, communication link slowness, a node being unable to sync with the blockchain, and many other scenarios.

## Secure off-chain computation

In some scenarios, it is desirable to be able to offload some intensive computation to off-chain systems; for example, if there is a requirement to do some computation that requires **High-Performance Computing** (**HPC**) resources.

This somewhat overlaps with integration, but it's mentioned here separately because of specific security requirements that the off-chain computations must be provably correct with integrity and authenticity guarantees.

### Better tools

Usually, in enterprise systems, there are many supporting tools and utilities packaged with the main product to operate the software. For example, these include administration tools, deployment tools, developer utilities, visualization tools, management tools, and end user tools. Blockchain platforms with better tooling are much more desirable because of better user support. Tools such as block explorers, user administration modules, and dApps to manage smart contracts are quite useful. They are gradually becoming more mature as the whole blockchain ecosystem is growing.

Now that we understand the features of enterprise blockchains, we will present a comparison between public and enterprise blockchains to help understand the main differences.

## Enterprise blockchain versus public blockchain

In this section, we'll provide a comparison between public and enterprise blockchains. Consider the table shown here, which assesses some points of comparison between the two blockchain types. This is not an exhaustive list; however, we'll touch on the most major points:

| Aspect | Public chains | Enterprise chains |
|---|---|---|
| Confidentiality | No. | Yes. |
| Anonymity | No. | Yes. |
| Membership | Permissionless. | Permissioned via voting, KYC, usually under an enterprise blockchain. |
| Identity | Anonymous. | Known users. |
| Consensus | PoW/PoS. | BFT. |
| Finality | Mostly probabilistic. | Requires immediate/instant finality. |
| Transaction speed | Slower. | Faster (usually, should be). |
| Scalability | Better. | Not very scalable, usually due to consensus choice. Usually, a much smaller number of nodes compared to public chains. |
| Regulatory compliance | Not usually required - difficult to achieve. | Required at times - comparatively easier to achieve. |
| Trust | Fully decentralized. | Semi-centralized and managed via consortium and voting mechanisms. |
| Smart contracts | Not strictly required; for example, in the Bitcoin chain. | Strictly required to support arbitrary business functions. |

The preceding table compares several key aspects of public and enterprise platforms. Next, we'll briefly consider some of the use cases of enterprise blockchains.

In the next section, we'll describe enterprise blockchain architecture, which can help us communicate with stakeholders, promote early design decisions, serve as a reusable model for development, build shared understanding, and understand how the system is structured.

# Enterprise blockchain architecture

A typical enterprise blockchain architecture contains several elements. We saw a generic blockchain architecture in *Chapter 1*, *Blockchain 101*, and we can expand and modify that a little bit to transform it into an enterprise blockchain architecture that highlights the core requirements of an enterprise blockchain. These requirements are mostly driven by enterprise needs and use cases:

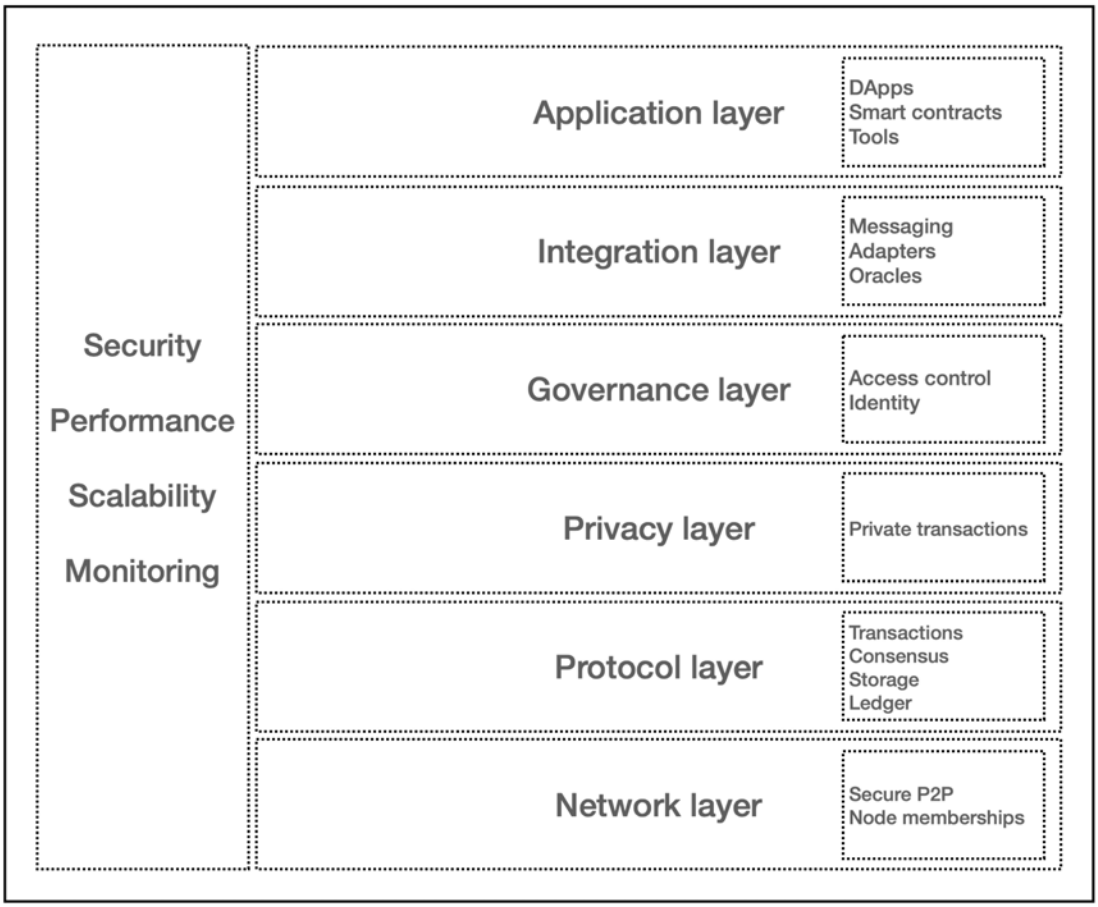



*Figure 16.1: Enterprise blockchain layered architecture*

We'll discuss each of these layers as follows:

- **Network layer**: The network layer is responsible for implementing network protocols such as **peer-to-peer** (**P2P**) protocols used for information dissemination.

- **Protocol layer**: This is the actual ledger layer, or blockchain layer, where the core consensus, transaction management, and storage elements are implemented.
- **Privacy layer**: This layer is responsible for providing one of the core features of enterprise blockchain: privacy. There are a number of ways to achieve privacy, including off-chain privacy managers, zero knowledge, and hardware-assisted privacy. Hardware-based privacy is usually supported using **trusted execution environments** (**TEEs**). On the other hand, software or algorithmic privacy such as ZKPs are also quite common in enterprise blockchains.
- **Governance layer**: This layer is responsible for providing the enterprise-grade access control mechanism that controls the consortium network membership. This can either be controlled via an on-chain permissioning system implemented in smart contracts, as part of the software client, or can be integrated with existing off-chain enterprise permissioning systems such as **Single Sign-On** (**SSO**) or **Active Directory** (**AD**).
- **Integration layer**: This layer provides APIs and a mechanism used to integrate with the legacy or existing back-office systems. This can be as simple as an RPC layer providing APIs over RPC or can constitute built-in connectors and plugins for integrating with the enterprise service bus.
- The integration layer is responsible for ensuring integration with back-office, legacy, and existing off-chain systems. It is not part of the core protocol but, as part of the holistic view of the blockchain end-to-end solution, this layer is vital for delivering business results. While there are many techniques available, a common integration framework used in the enterprise environment is **Apache Camel**. This can also be used in blockchain solutions as it comes with the Ethereum Web3J library component.

Apache Camel is an open source enterprise integration framework. It enables easy integration between various systems by utilizing enterprise integration patterns. It has hundreds of components for different systems such as databases, APIs, and MQs for easy integration between systems.

For example, the Web3J connector (Apache Camel Ethereum connector) is a feature-rich connector available in Apache Camel that enables integration between Ethereum chains and other systems. More details on the Web3J component are available here: `https://camel.apache.org/components/latest/web3j-component.html`.

The Apache Camel Ethereum connector works with Ethereum Geth nodes, Quorum, Parity, and Ganache. It supports JSON RPC API over HTTP and IPC with implementations for different blockchain operations such as `net`, `eth`, `shh`, and so on. It has support for Ethereum filters, **Ethereum Name Service** (**ENS**), and JSON RPC API, and is also a fully tested (unit and integration) solution.

A generic high-level design using Apache Camel is shown in the following diagram:

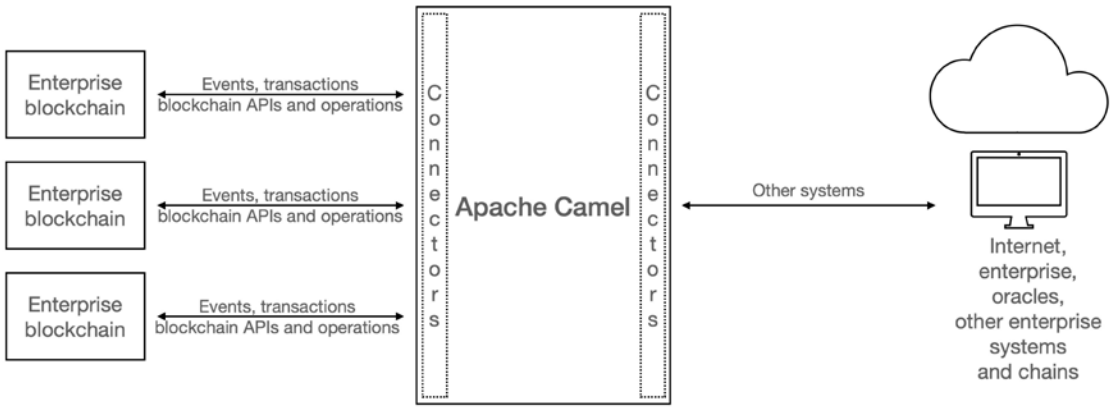



*Figure 16.2: High-level design of Apache Camel, which enables blockchain integration*

In the preceding diagram, we can see several enterprise blockchains connecting to Apache Camel using Apache Camel Ethereum connectors. Also, other systems connect to Apache Camel via their respective connectors. Apache Camel via connectors is responsible for integrating all these systems. For example, it is entirely possible to use Apache Camel to take data from enterprise blockchains and store it in traditional SQL databases in the enterprise via suitable connectors.

- **Application layer**: The application layer, as the name suggests, consists of **dApps**, **smart contracts**, tools, and other relevant software to support enterprise use cases.
- **Security, performance, scalability, monitoring**: On the left-hand side of the architecture diagram (*Figure 16.1*), security, scalability, performance, and monitoring are shown. As each layer in the enterprise blockchain benefits from (and indeed, requires) security, scalability, and performance, this layer is shown as encompassing all layers. Monitoring also plays a vital role in any enterprise solution. Without effective visualizations in such complex networks, it is almost impossible to keep track of everything. Therefore, this layer is shown as relevant to all enterprise blockchain layers.

Now that we understand the architecture of enterprise blockchains in general, let's dive a little bit deeper into the mechanics of designing enterprise blockchain solutions. In the next section, we'll see which tools and frameworks we can use to build enterprise solutions.

# Designing enterprise blockchain solutions

An isolated blockchain in an enterprise is not sufficient to solve business problems. In addition to choosing a blockchain platform, there are some other factors to consider while introducing a blockchain in an enterprise. On top of the list of these factors is integration with the existing back-office and legacy systems.

Without any architecture framework, it becomes quite challenging to have a holistic view of an enterprise and see how all business processes fit together. As such, a question arises regarding whether we can leverage existing frameworks to address enterprise blockchain architectural needs. The answer is *yes*. There are already established and mature frameworks to facilitate enterprise architecture development.

The purpose of enterprise architecture frameworks is to enable an organization to execute its business strategy effectively. It allows an organization to see an organization from different perspectives, including business, information, process, and technology, and make effective business decisions to achieve business goals.

Let's now explore the popular enterprise architecture frameworks, **The Open Group Architecture Framework** (**TOGAF**) and Zachman Framework.

## TOGAF

**TOGAF** stands for **The Open Group Architecture Framework**. It is developed by the Open Group.

The official TOGAF website can be found here: `https://www.opengroup.org/togaf`.

It is a framework that enables organizations to systematically design, plan, and implement enterprise solutions in businesses. It has four architectural domains:

- Business architecture domain: This domain defines the business strategy, organization structure, business processes, and governance of the organization.
- Data architecture domain: This domain describes the structures of an organization's data assets and relevant data management resources.
- Application architecture domain: This domain describes the enterprise applications, deployment blueprints, relationships, and interactions between applications, along with the relationships these applications have with business processes within the enterprise.
- Technology architecture domain: This domain defines the requirements of the technical architecture implementation of the enterprise applications. This includes the description of hardware, software, middleware, and network infrastructure required for the implementation of the enterprise applications.
- Now, re-examining each domain with blockchain in mind, we can include blockchain at each layer and make decisions such as business requirements, the application architecture, logical and physical data assets, and finally, the infrastructure required for implementing the enterprise blockchain solution.

After this basic introduction to TOGAF, let's dive a bit deeper to understand the method TOGAF uses to develop an IT architecture.

## Architecture development method (ADM)

TOGAF ADM is a method for developing IT architecture that meets organizational business needs. It describes the process of moving from the foundational TOGAF architecture to an organization-specific architecture. This is an iterative process with continuous requirements management, which results in the development of an architecture that is specific to an organization and addresses specific needs. Once the architecture development is complete, the architecture can be published throughout the organization to develop a common understanding.

The ADM is a tested and repeatable process for developing enterprise architectures. The process follows the following steps: establish an architecture framework, develop architecture content, architecture governance, and implementation of architectures.

The ADM has nine phases and each phase can be further divided into multiple steps. The ADM model is shown here:

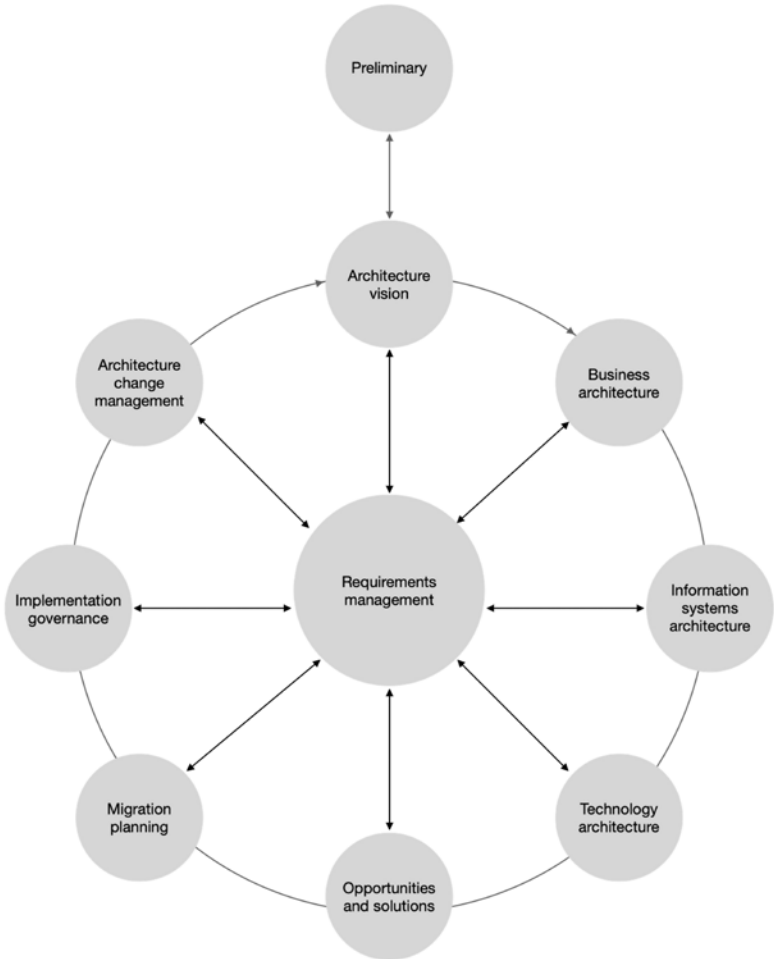



*Figure 16.3: ADM model*

The original diagram can be found at `https://pubs.opengroup.org/architecture/togaf9-doc/arch/chap05.html`.

Now, we'll describe each phase in more detail:

- Preliminary phase: This phase defines the groundwork of the architecture. It defines methodologies, architecture principles, the architecture scope, and assumptions about the architecture. It also defines the responsible parties for architecture delivery. From a blockchain perspective, we can define the overall blockchain strategy in this phase.
- Architecture vision: This phase validates business principles, goals, and business strategy. It defines key business requirements, as well as a high-level description of the business value expected from the architecture work. This phase also analyzes the impact of the new architecture on other processes.
- Business architecture: This phase proposes the baseline business architecture and develops a target business architecture, along with gap analysis. In a blockchain scenario, we can define the current state architecture and target state architecture at this stage.
- Information systems architecture: This phase defines the data architecture and application architecture. The data architecture includes building the baseline data architecture, the data architecture description, building data architecture models, doing impact analysis review reference models, and viewpoints. Application architecture defines the baseline application architecture, builds application architecture models, and proposes applications. Both of these activities also perform gap analysis to validate the architecture being developed, as well as to find any shortfalls between the baseline architecture and the target architecture. Similar to the business architecture, for an enterprise blockchain solution, we can define the current state architecture and target state architecture.
- Technology architecture: This phase is primarily concerned with reviewing the baseline business, data, and application architecture and building a baseline description of the current technology architecture in the enterprise. It also proposes the target technology architecture. In this phase, we can propose a target enterprise blockchain platform and the target solution.
- Opportunities and solutions: This phase performs evaluations and selects various proposed target architectures. Also, an implementation strategy and plan are proposed at this stage. We can apply this to blockchain in a similar fashion, by evaluating or selecting the target architecture encompassing enterprise blockchain solutions. When evaluating blockchain solutions, a number of features that we presented earlier in the comparison between enterprise blockchain and public blockchain can also be used; for example, confidentiality, scalability, and finality.
- Migration planning: This phase creates and finalizes the comprehensive implementation plan for migrating to the target architecture from the current architecture.

- Implementation governance: This phase deals with the implementation of the target architecture. A strategy to govern the overall deployment and migration is developed here. We can also perform blockchain solution testing and devise a deployment strategy during this phase.
- Architecture change management: This phase is responsible for creating change management guidelines and procedures for the newly implemented target architecture. From a blockchain perspective, this phase can provide change management procedures for the newly implemented enterprise blockchain solution.

With this, we've completed our introduction to TOGAF and explored the idea that enterprise blockchain solutions should be viewed through the lens of enterprise architecture.

The fundamental idea to understand here is that enterprise blockchain solutions are not merely a matter of quickly spinning up a network, creating a few smart contracts, creating a web frontend, and hoping that it will solve business problems. This setup may be useful as a PoC or for an incredibly simple use case but is certainly not an enterprise solution. We suggest that an enterprise blockchain solution must be looked at through the lens of the enterprise architecture and regarded as a full-grade enterprise solution. This is so that we can effectively achieve the business goals intended to be solved by enterprise blockchain solutions.

Next, let's explore how we could implement a blockchain business solution in an organization that has moved its operations to the cloud.

## Blockchain in the cloud

Cloud computing provides excellent benefits to enterprises, including efficiency, cost reduction, scalability, high availability, and security. Cloud computing delivers computing services such as infrastructure, servers, databases, and software over the internet. There are different types of cloud services available; a standard comparison is made between **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**). A question arises here: where does blockchain fit in?

**Blockchain as a Service**, or **BaaS**, is an extension of SaaS, whereby a blockchain platform is implemented in the cloud for an organization. The organization manages its applications on the blockchain, and the rest of the software management, infrastructure management, and other aspects, such as security and operating systems, are managed by the cloud provider. This means that the blockchain's software and infrastructure are provided and maintained by the cloud provider. The customer or enterprise can focus on their business applications without worrying about other aspects of the infrastructure.

The following is a comparison of different approaches:

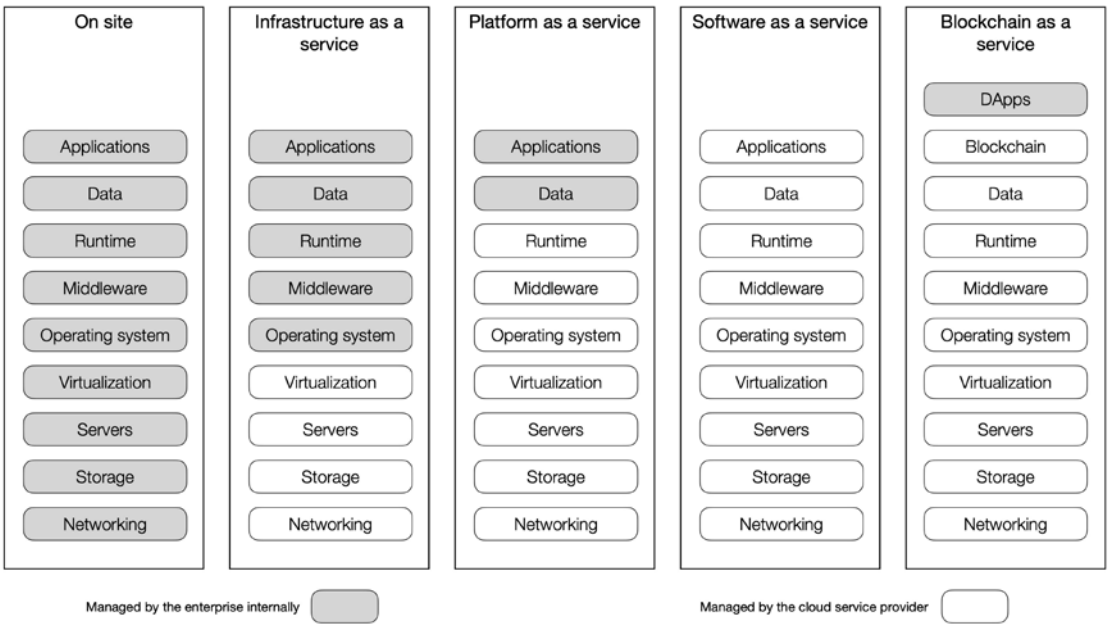



*Figure 16.04: Cloud solutions*

BaaS can be thought of as a type of SaaS, where the software is a blockchain. In this case, just like in SaaS, all services are externally managed. In other words, customers get a fully managed blockchain network on which they can build and manage their own dApps. Note that in the preceding diagram, under the **Blockchain as a Service** column, **Applications** have been replaced with **Blockchain**, as a differentiator between other cloud services and BaaS. Here, blockchain is the software (application) provided and managed by the cloud service provider. Also, note that dApps have been added on top, which are managed by the enterprise internally.

There are many BaaS providers. A few of them are listed as follows:

- **AWS**: `https://aws.amazon.com/blockchain/`
- **Azure**: `https://azure.microsoft.com/en-gb/solutions/blockchain/`
- **Oracle**: `https://www.oracle.com/uk/application-development/cloud-services/blockchain-platform/`
- **IBM**: `https://www.ibm.com/uk-en/cloud/blockchain-platform`

In the next section, we'll introduce some enterprise blockchain platforms.

## Currently available enterprise blockchains

As this is a very ripe area for research, and indeed the market is very active in this space, there have been several enterprise blockchain solutions developed and made available in the last few years. Notably, the year 2019 has been called "the year of the enterprise blockchain" as many startups and enterprises focused on this area emerged.

In this section, we will not cover all these platforms in detail but will provide a brief introduction and links to more details on these chains:

- **Quorum** is an open source enterprise blockchain platform. It aims to support enterprise business needs and allows a business to achieve business goals (and unlock economic value) by leveraging blockchain technology. It addresses crucial enterprise requirements including privacy, performance, and enterprise permissioning. We will also explore Quorum in greater detail later in this chapter.
- **Fabric** is a **Hyperledger** project. It is an enterprise-grade distributed ledger that allows the development of blockchain solutions with a modular architecture. It has a permissioned architecture, support modularization, and pluggable consensus, and supports smart contracts. We discussed Hyperledger and other projects under it, such as **Sawtooth**, in detail in *Chapter 14, Hyperledger*.

Let's consider a comparison of leading enterprise blockchain platforms. We'll cover most of the desirable features of enterprise blockchains. This can be used as a reference for a quick comparison between these platforms. This comparison is based on the current implementations of these platforms available at the time of writing. However, as this is a very rapidly changing area, some features may change over time, or new features might be added or improved:

| Feature | Quorum | Fabric | Corda |
|---|---|---|---|
| Target industry | Cross-industry | Cross-industry | Cross-industry |
| Performance (approximate transactions per second, or TPS) | 700 (*) | 560 (∞) | 600 (@) |
| Consensus mechanism | Pluggable multiple Raft, IBFT, PoA | Pluggable Raft | Pluggable, notary-based |
| Tooling | Rich enterprise tooling | SDKs | Rich enterprise tooling |
| Smart contract language | Solidity | Golang | Kotlin/Java |
| Finality | Immediate | Immediate | Immediate |
| Privacy | Yes (restricted private transactions) | Yes (restricted private transactions) | Yes (restricted private transactions) |

| | | | |
|---|---|---|---|
| Access control | Enterprise-grade permissioning mechanism | Membership service providers/certificate based | Doorman service/KYC. Certificate-based |
| Implementation language | Golang, Java | Golang | Kotlin |
| Node membership | Smart contract and node software managed | Via membership service provider | Node software managed using configuration files, certificate authority controlled |
| Member identification | Public keys/ addresses | PKI-based via membership service provider, supports organization identity | PKI-based, support organization identity |
| Cryptography used | `SECP256K1`<br>`AES`<br>`CURVE25519 + XSALSA20 + POLY13050`<br>`PBKDF2`<br>`SCRYPT` | `SECP256R1` | `ED255519`<br>`SECP256R1`<br>`RSA – PKCS1` |
| Smart contract runtime | EVM | Sandboxed in Docker containers | Deterministic JVM |
| Upgradeable smart contract | Possible with some patterns, not inherently supported | Allowed via upgrade transactions | Allowed via administrator privileges and auto-update allowed under administrative checks |
| Tokenization support | Flexible—inherited from public Ethereum standards | Programmable | Corda token SDK |

* TPS results for Quorum are based on `https://arxiv.org/pdf/1809.03421.pdf`.

∞ TPS results for Hyperledger Fabric are based on `https://hyperledger.github.io/caliper-benchmarks/fabric/performance/2.0.0/nodeContract/nodeSDK/submit/empty-contract/`.

@ TPS results for Corda are based on `https://www.r3.com/corda-enterprise/`.

This comparison may also be used for the quick evaluation of these platforms and their suitability for an enterprise use case.

Now that we've examined some requirements, architectures, and use cases of enterprise blockchains, let's briefly describe some of the challenges that enterprise blockchains face. These can be attributed to public chains as well, but enterprise chains suffer from these specific issues too.

# Enterprise blockchain challenges

While enterprise blockchains have addressed core enterprise requirements (privacy, performance, and governance) to some extent, there are still more challenges that need to be addressed. There is significant progress being made toward solving these issues. However, there is still a lot of work required to be done to adequately address these limitations. Some of these limitations are listed as follows.

## Interoperability

Blockchain solutions are built by different development teams with different target requirements. As a result, there are now many different types of blockchains, ranging from cryptocurrency public blockchains to application-specific blockchains, which are developed for a single business application. Data exchange between these chains is a crucial concern. While blockchain networks continue to grow independently, integration and interoperability between these chains remain a big concern.

This problem also stems from a lack of standardization. If a standard specification is available, then all chains following that standard will become compatible automatically. However, what about those chains that are already deployed in production, including public chains? How do we achieve interoperability between them? Many business use cases have the requirement to get data from one organization to another or from one blockchain network to another. This is also true in the case of data exchange requirements between a public blockchain and a consortium network.

## Lack of standardization

Lack of standardization is a commonly highlighted concern in enterprise blockchains and generally in the blockchain ecosystem. Traditional systems are mostly developed in line with standards defined by standards bodies, such as NIST FISMA standards for information security.

The **Federal Information Security Management Act** (**FISMA**) of 2002 is a United States federal law that requires all federal agencies to develop, document, and implement an agency-wide information security program. FISMA was amended in 2014 as the Federal Information Security Modernization Act. More on FISMA here: `https://www.congress.gov/bill/113th-congress/senate-bill/2521`.

Standards are also essential to achieving interoperability. Several initiatives have been taken to address the challenges mentioned here and also to standardize enterprise blockchain platforms. A leading organization in this field is EEA, a standards organization run by its members that aims to develop enterprise blockchain specifications.

EEA regularly releases technical specifications that can be downloaded at the link provided here:

`https://entethalliance.org/technical-specifications/`.

EEA's official website is `https://entethalliance.org`.

## Compliance

There are various compliance requirements in almost all industries—especially finance, law, and health—where strict guidelines and rules have been mandated by regulatory authorities. Enterprise systems are expected to conform to these requirements. Blockchain initially started with a different focus of building an electronic cash system with an egalitarian philosophy. Still, surely enterprises have different visions and mindsets. Some of the regulations include GDPR, SOX, and PCI, which define different requirements for an enterprise to conform to. Also, compliance with technical standards such as the NIST FISMA standard for information security is necessary.

There are also jurisdiction issues. A distributed network with geographically dispersed locations may require different legal requirements to be met in various jurisdictions. For example, a specific type of cryptography may be allowed by law in the US but not in Cuba.

## Business challenges

From a business perspective, cost, funding, and governance are some of the challenges that need to be met. If building and implementing an enterprise blockchain solution is prohibitively costly, then the business stakeholder might prefer traditional enterprise systems. Also, from an operational perspective, training staff to operate blockchain solutions might be a concern. We can categorize all blockchain-related costing, funding, and economics under an umbrella term that we call **Enterprise Blockonomics**. Enterprise Blockonomics (blockchain economics) can be defined as a study of cost and cost models in the context of enterprise blockchains.

With that, we have covered a lot of background material and developed an understanding of the enterprise blockchain requirements, architecture, and challenges. Let's now dive into some examples of enterprise blockchain platforms. First, we'll discuss VMware, and then we'll cover Quorum, a popular enterprise blockchain platform, in detail.

A recent addition to the enterprise blockchain world is **VMware Blockchain** (**VMBC**), which we introduce next.

# VMware Blockchain

VMBC is an enterprise-grade blockchain platform that provides features suitable for enterprise use cases. It has introduced many innovations that result in better performance, flexibility, security, and scalability of blockchain dApps.

## Components

The core components of VMware Blockchain are:

- Modular framework – a modular framework such that various smart contract execution engines (such as DAMLe or EVM) can be used. DAMLe, or the DAM execution engine, basically converts application layer actions into events on distributed ledger layer.
- Replica network – a set of replica nodes that present a single and consistent state machine to the client nodes.
- Replica nodes:
    - Replica nodes use a consensus mechanism to implement the SBFT state machine replication protocol, which forms a replica network composed of replica nodes.
    - Each replica node has a unique identifier, stores a replicated state machine, and makes use of the local key-value database (RocksDB). One replica is selected as a leader during a consensus cycle. Moreover, some replica nodes have collector roles. There are two types of collectors: C-collector and E-collector. C-collector collects commit messages and a combined signature is sent back to replicas to confirm the commit. E-collector collects all execution messages, and a combined signature is sent back to replicas and clients as a proof certificate of their request's execution.
- Group of client nodes:
    - Regular client nodes – provide an interface between applications and the replica network. Client nodes incorporate the BFT client and application interfaces such as the DAML ledger API server.
    - Full copy client node – receives all the updates in contrast to the regular client node, which receives a filtered view of the data it is permitted to access.
    - Duplicate client node – used for providing high availability and load balancing.
- VMWare Blockchain Orchestrator – a standalone virtual appliance that deploys on-premises VMware Blockchain nodes.
- There are also tools available for node deployment and monitoring and fleet management.
- An object store is used for archiving the data.

VMware, as with other blockchains, relies on a consensus protocol to ensure agreement among its participants, which we discuss next.

## Consensus protocol

Consensus in VMware Blockchain is a PBFT variant, called SBFT. It is more scalable and decentralized as compared to previous PBFT variants. It guarantees safety, liveness, and linearity. Linearity, in simple words, means constant size messages, which result in a constant cost of operations on the chain. The leader election mechanism in SBFT ensures that a different group of collectors is chosen for each block, which spreads the load of primary leadership and collectors among all replicas. SBFT has reported roughly around 70 transactions per second on a geographically dispersed heavy-load network.

There are three core improvements including:

- Reduced communication because of using collectors, which enables linear communication and avoids quadratic all-to-all communication. Moreover, a single confirmation is sent to the clients.
- Implementation of faster and smaller sized (33 bytes) BLS signatures, which reduces proof length. Threshold signatures also reduce client communication from linear to constant.
- One round optimistic path consensus, which is important for reducing latency and results in faster agreement. There is also a faster optimistic agreement path for executions in normal cases.

With all the components we introduced so far, we can now describe the architecture of VMware Blockchain.

## Architecture

We can visualize the high-level architecture of VMware Blockchain in the diagram shown below:

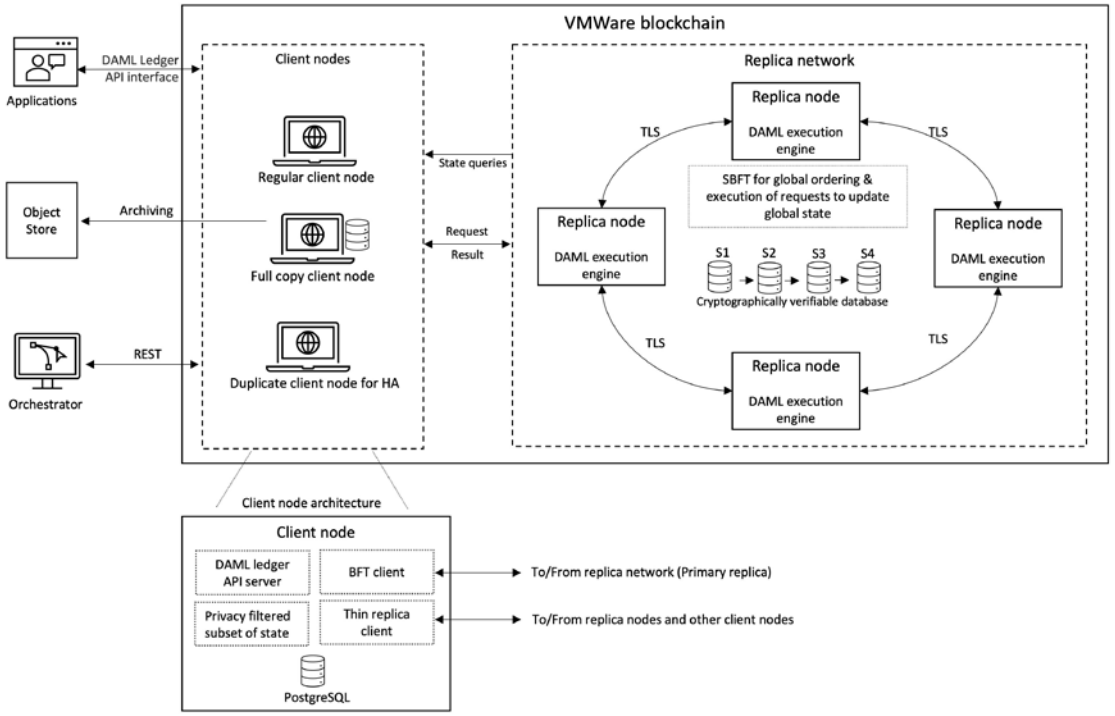



*Figure 16.5: VMware Blockchain high-level architecture*

The diagram shows a replica network composed of four nodes and a cryptographically verifiable database. The final state is achieved after running the SBFT consensus algorithm. The client nodes communicate with the replica network via TLS-secured communication channels. Client nodes (shown on the left side of the diagram) can make state queries and request information form the primary replica node in the replica network.

A client node consists of a DAML ledger API server and a privacy-filtered subset of state, as a node only sees what it is authorized to see. The node also consists of a BFT client that speaks to the primary replica in the replica network, and a thin replica client, which runs the thin replica protocol.

The client node accepts requests coming from applications and sends them to replicas, receives results from the replica network, and sends them back to the applications. The dApp (application) sends commands to the DAML ledger API server component of the client node. The DAML API components forward the command to a pool of BFT clients, which sends the requests to the replica network for execution and waits to collect the results from the replica nodes. The BFT client collects and ensures the validity of the reply (results) it receives from the replica network and forwards the results back to the DAML ledger API. The DAML ledger API receives the results of the execution using the PostgreSQL notification. Using the thin replica protocol, a replica node sends the results to other client nodes that are subscribed to receive these updates. The client node using the thin replica protocol receives the update and writes it into the local client PostgreSQL database. A notification indicating the update in the local DB is sent to the DAML ledger API. The Postgres SQL DB stores a materialized view of the replica network, which is the data that the specific client node is permitted to view in a processed format. Each replica node contains a **replicated state machine** (**RSM**), authenticated local key-value store (RocksDB), and DAMLe – DAML execution engine. The database is cryptographically verifiable. It supports checkpoints for archiving. Request ordering is achieved via consensus.

Other components of the blockchain include applications that talk to the client nodes via the DAML ledger API interface. An object store also exists, which is used for archiving data via a full copy client node. Finally, an orchestrator appliance is used to create, deploy, and manage VMware Blockchain via a REST interface.

VMBC supports DAML and EVM. Parallel execution is supported by the DAML engine.

More information on this blockchain is available here: `https://www.vmware.com/uk/products/blockchain.html`

DAML is a statically typed functional smart contract language – a DSL, influenced by Haskell. It is a concise, English-like, easy-to-understand and write, business-oriented language. It is designed to build a multi-party composable application. It is also called "Enterprise Haskell." It is basically Haskell, with added primitives for smart contracts, authorization rules, and privacy. DAML aims to solve portability, interoperability, and privacy problems in the blockchain. It focuses on data privacy and the authorization of distributed applications. In this language, parties are first-class primitives. It is built with privacy in mind and enables tracking and authorization at each workflow step – i.e., each smart contract has its own defined privacy. By abstracting data privacy and authorization, DAML allows developers to focus on workflow logic instead of worrying about concrete cryptographic primitives. It is also storage layer agnostic, portable, and interoperable, which means that DAML code written for one platform will work for other platforms too, without any change. For example, the same DAML code will work for both Hyperledger and Ethereum, and even a SQL database without any change.

You can find out more about DAML here: `https://www.digitalasset.com/developers`. Note that DAML is developed by Digital Asset Holdings.

## VMware Blockchain for Ethereum

Public Ethereum is unsuitable for enterprise use cases due to scalability, strict governance, enterprise-grade operational support, and privacy requirements. Nevertheless, it is the most used platform for smart contract development due to its mature development ecosystem. It can become suitable for enterprise use cases if we can address the limitations in the public Ethereum chain. VMware Blockchain for Ethereum addresses these issues and provides a platform that allows easy development, efficient operations, and scalable Ethereum blockchain networks. Ethereum support can be provided in addition to DAML support already available in VMware Blockchain. VMware Blockchain for Ethereum is a high-performance, high-throughput platform with ZKP-based programmable privacy capabilities, enterprise-grade operations, and governance features. Developers can use familiar tools like MetaMask, Remix, Hardhat, and Truffle to develop on this platform. Moreover, it supports standard Ethereum APIs providing a seamless developer experience.

In summary, we can say that with all the innovations, improvements, and smart choices of enterprise-friendly features, VMware Blockchain is a worthy choice for enterprise use cases.

# Quorum

**Quorum** is an open source enterprise blockchain platform. It is a lightweight fork of **Ethereum**. Quorum not only benefits from the innovation and research being done in the upstream public Ethereum (Geth) project but also many excellent enterprise features have been introduced in Quorum. These enterprise features primarily focus on providing enterprise-grade privacy, performance, and permissioning (access control).

First, let's take a look at Quorum's architecture.

## Architecture

Quorum addresses three fundamental issues in public blockchains, which makes it an excellent choice for enterprise use cases:

- Privacy
- Performance
- Enterprise governance

At a high level, the Quorum architecture consists of nodes and their associated privacy managers. The Quorum architecture can be visualized using the following diagram:

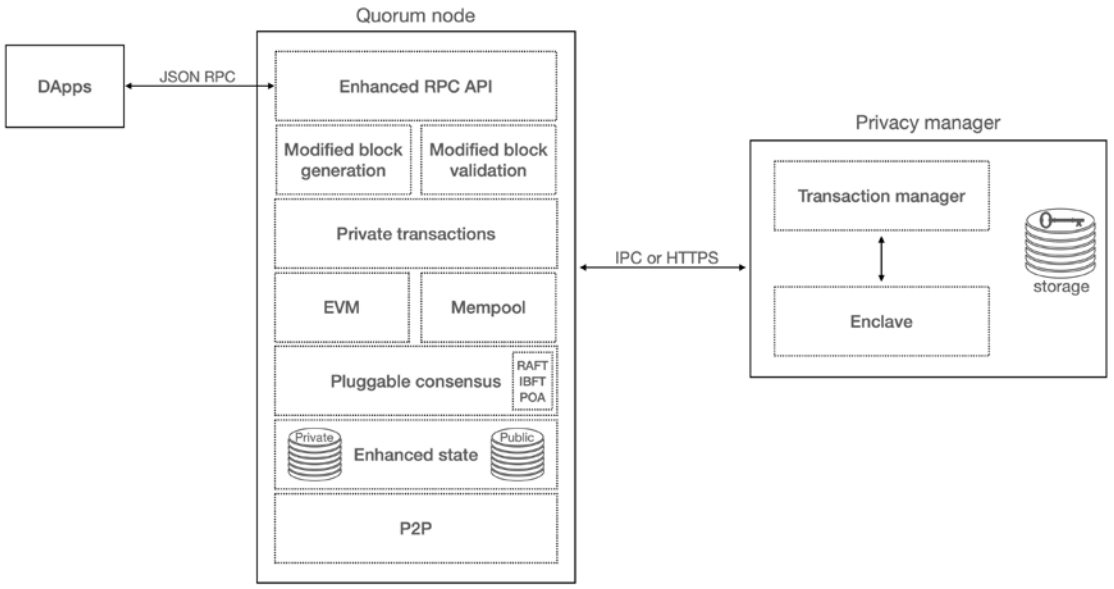



*Figure 16.6: Quorum architecture*

## Nodes

The Quorum node is a modified and enhanced version of public `geth` that supports private transactions. Quorum nodes communicate via HTTPS with privacy managers, which, in turn, are responsible for providing privacy by storing the payload in encrypted format in their local storage. These Quorum nodes maintain the public and private states separately.

There are several changes that have been made to the Quorum client to make it suitable for enterprise use cases. We'll provide an overview of each of those changes here:

- **Enhanced P2P**: This layer is modified to allow connections only between authorized nodes.
- **Enhanced state (private and public)**: In addition to the public state trie, there is an additional Merkle Patricia state trie for the private state. In Ethereum, a modified **Merkle Patricia trie** (**MPT**) is a data structure that is used to provide a cryptographically authenticated key-value store. More technical details on this are available here: `https://eth.wiki/en/fundamentals/patricia-tree`.
- **Pluggable consensus**: Quorum supports multiple consensus algorithms such as IBFT, **Proof of Authority** (**PoA**), and Raft. Quorum, being an enterprise blockchain, does not need a PoW type of consensus.
- **No transaction fees**: The pricing of gas has been set to zero as there is no need for transaction fees in consortium networks.

- **Private transactions**: Private transactions are identified using `37` or `38` as the `V` value in the transaction. The transaction creation mechanism is modified to enable replacing transaction data (input) with the hash of the encrypted payload.
- Remember that in standard Ethereum, as we learned in *Chapter 9*, *Ethereum Architecture*, a transaction's `V` value can be either `27` or `28`, but in Quorum, that has been changed to `37` or `38` to indicate private transactions. Public transactions are still identifiable by a `27` or `28` `V` value, whereas private transactions are identifiable with a `V` value of either `37` or `38`.
- **Modified block generation mechanism:** The logic for generating blocks is modified with a new check for the global public state root instead of the global state root.
- **Modified block validation mechanism**: Block validation logic is modified to replace the global state root check with a global state root check *for public state*—otherwise known as a global public state root.
- **Enhanced RPC API:** Quorum supports additional RPC APIs that help us interact with the enhanced enterprise features of Quorum, such as permissioning and consensus mechanisms.

Now, we'll discuss **privacy manager**, which consists of two components: a transaction manager and an enclave.

## Privacy manager

**Privacy manager** is an off-chain mechanism that provides transaction confidentiality. It is paired on a one-to-one basis with a Quorum node. It allows Quorum nodes to share the transaction payload securely between authorized participants.

The **transaction manager** is responsible for:

- Storing encrypted transaction payloads
- Managing access to encrypted transaction payloads
- Propagating encrypted payloads to other transaction managers on the network
- Discovering other transaction managers on the network

Quorum has developed two transaction managers. Initially, a Haskell implementation called **Constellation** was made available. However, it is no longer actively developed in favor of a more feature-rich, Java-based privacy manager called **Tessera**. Tessera is developed in Java. It is used for the storage, encryption, decryption, and propagation of private transaction data.

An **enclave** is an isolated and independent element that provides cryptography services for transaction payload encryption and decryption. It can only be associated on a one-to-one basis with its own transaction manager.

In order to achieve all the desired privacy features, several cryptographic protocols and primitives have been used in the Quorum platform. We'll provide a quick overview of these in the next section.

## Cryptography

Quorum inherits its cryptography stack from public **Ethereum.** It includes standard ECDSA signatures, AES CTR cryptography for wallets and DEVP2P, and Keccak256 hash functions. Privacy manager makes use of Curve25519, **Elliptic-curve Diffie-Hellman** (**ECDH**), poly1305, and Xsalsa20 cryptographic operations to provide confidentiality guarantees required by private transactions in Quorum.

Now, we'll explore how each of the main facets—privacy (confidentiality), enterprise-grade membership control (access control, and the permissioning mechanism), and performance—are achieved in Quorum.

## Privacy

Quorum supports both private and public transactions. For private transactions, it uses a mechanism called private transaction manager, which is an off-chain component to facilitate the confidentiality of transactions. We will now describe how private transactions work in Quorum.

Suppose there are three parties: *A*, *B*, and *C*. Parties *A* and *B* are privy to a transaction, while party *C* is not. We'll now explore how the private transaction is generated and flows between parties and is propagated on the network while maintaining confidentiality. Let's call this transaction *transaction AB*:

1. To begin private *transaction AB*, party *A* creates a transaction and signs it before sending it to their Quorum node, node *A*. The transaction is composed of a transaction payload and the public key of the intended recipient. This list of public keys of intended recipients is maintained in the `PrivateFor` list. It can be a single public key or multiple keys, depending on the requirements.
2. There are two signing mechanisms in Quorum. For public transactions, an EIP55-based mechanism is used, and for private transactions, an Ethereum Homestead signing mechanism is used. Transactions in Quorum can also be signed independently, without using Quorum's signing mechanism.
3. Quorum node *A* sends the transaction to transaction manager *A* for processing.
4. Transaction manager *A* makes an encryption request to its enclave, to encrypt the transaction payload.
5. Party *A*'s enclave encrypts the transaction payload and sends it to transaction manager *A*.
6. Transaction manager *A* stores the transaction payload and sends it to other transaction managers, in this case, *B*.
7. A transaction hash is returned to Quorum node *A* by transaction manager *A*, which replaces the original transaction payload with the hash and changes the value `V` of the transaction to `37` or `38` to indicate that the transaction is private.
8. The transaction propagates to other nodes via the normal Ethereum P2P protocol.
9. The block that contains *transaction AB* finalizes and propagates on the network between all nodes (*A*, *B*, and *C*).
10. When the block containing the transaction is received by the Quorum nodes, they recognize the transaction as private because the `V` value of the transaction is either `37` or `38`.

If the transaction is identified as private, the Quorum nodes will query their respective transaction managers to figure out if they are party to the transaction or not. This is achieved by finding out if there is an entry available in the database with the transaction hash. Here, parties *A* and *B* will have the transaction hash stored in the transaction manager, whereas party *C* will not.

11. The transaction managers of parties *A* and *B* make a transaction payload decryption request to their respective enclaves.
12. The enclaves of parties *A* and *B* decrypt the private transactions.
13. When the transaction managers receive the decrypted payload back from their enclaves, they send this data to their respective Quorum nodes. In our example, the transaction managers of parties *A* and *B* will send the transaction payload to their respective Quorum nodes, *A* and *B*. The Quorum nodes will execute the transaction (contract) via the EVM and update their private state database accordingly. The transaction manager of party *C* will return a message to its Quorum node, indicating that it is not privy to (or a recipient of) the transaction. This means that the Quorum node of party *C* will simply ignore the transaction.

This process can be visualized using the following diagram:

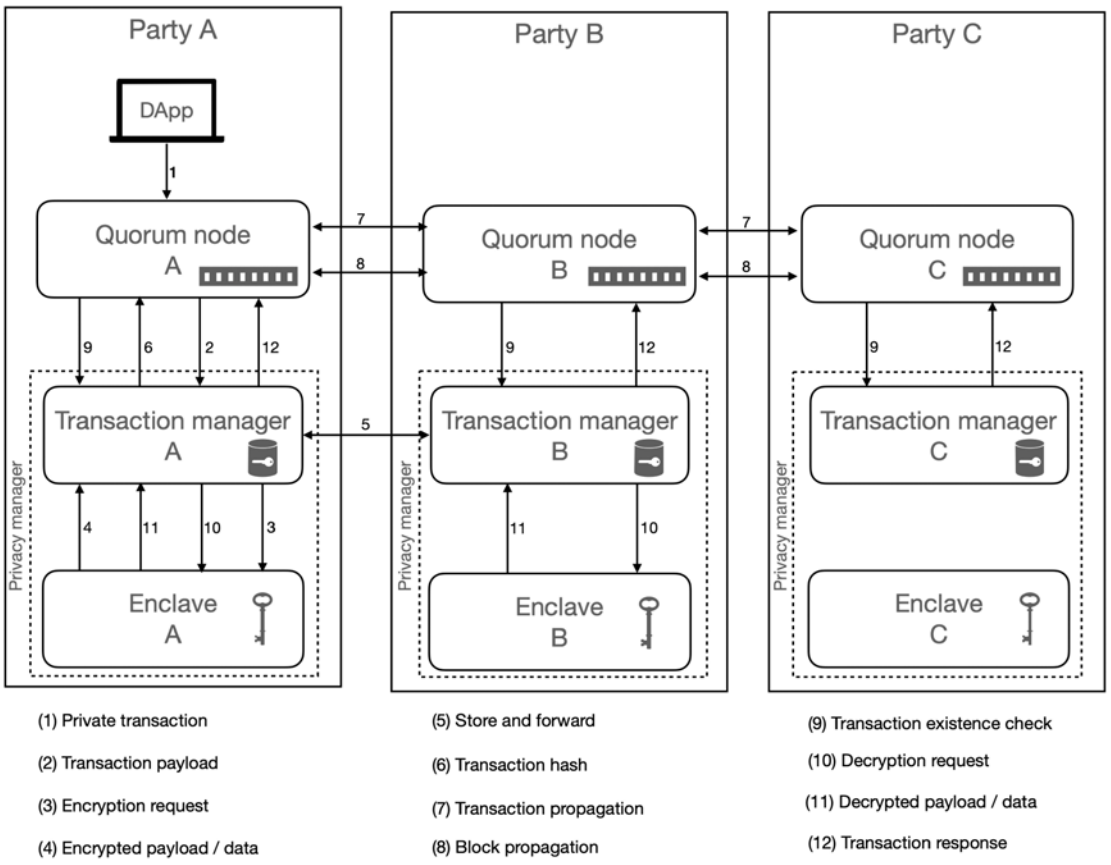



*Figure 16.7: Quorum private transaction flow*

Let's now expand on *Step 4* from the *achieving privacy* process, enclave encryption, as it consists of several sub-steps that are performed within the enclave.

## Enclave encryption

When an **enclave** receives a transaction, it performs several steps to encrypt the transaction. All these steps are shown here and are performed by the enclave as part of the transaction payload encryption process:

1. Generates a symmetric key for the transaction.
2. Generates two random nonces.
3. Encrypts the transaction payload and one of the nonces with the symmetric key generated in the first step.
4. Encrypts the transaction key for transactions for each recipient. For this purpose, the following steps are performed:
    - Generates a shared symmetric key by utilizing ECDH. ECDH will use the sender's private key and the receiver's public key.
    - Encrypts the symmetric key for transaction for each receiver separately using the newly generated shared symmetric key and the second nonce.
    - This step will be repeated for each recipient.
5. Finally, the transaction manager receives the encrypted transaction payload, all the encrypted symmetric keys for transactions, both nonces, and the public keys of the senders and receivers.

Now, let's see how the transaction manager stores and propagates transactions to the other transaction manager. We'll now expand on *Step 5* of the *achieving privacy* process mentioned previously.

## Transaction propagation to transaction managers

A **transaction manager** performs the following steps after it receives an encrypted response back from the **enclave**:

1. It generates the SHA3-512 hash of the encrypted payload received from the enclave.
2. It stores the encrypted payload, the hash and the encrypted symmetric key for the transaction, both nonces, and the public keys of the sender and receiver.

3. It sends, using HTTPS, the encrypted payload, the hash, and the encrypted symmetric key for the transaction using transaction manager *B*'s public key to transaction manager *B*.
4. Transaction manager *A* awaits an acknowledge message from transaction manager *B*. If an acknowledgment is not received, the transaction will not be propagated to the network.

Let's now discover how the decryption process works in enclaves. This corresponds to *Step 11* of the *achieving privacy* process.

## Enclave decryption

The transaction payload decryption process starts when a transaction payload decryption request is made to an **enclave**. The enclave performs the following steps to decrypt a payload:

1. Derives the shared symmetric key. The key derivation process works as follows:
    - Party *A*, being the sender of the transaction, derives the shared symmetric key using its private key and the receiver's public key.
    - Party *B*, being the recipient of the transaction, derives the shared symmetric key using its private key and the sender's public key.
2. Decrypts the symmetric key for transactions with the shared symmetric key, along with the encrypted payload and nonce, fetched from the database.
3. Decrypts the transaction data with the symmetric key for transactions and the encrypted data and nonce fetched from the database.
4. Finally, the decrypted private transaction data is sent to the transaction manager.

The enclave, being separated from the transaction manager, addresses the **separation of concerns** and allows parallelization in order to improve performance. Separation of concerns is a form of abstraction that allows the segregation of different components of computer software into separate and distinct units so that each unit addresses a separate concern. This simplifies design and helps to modularize the software.

The overall process (enclave encryption, transaction manager storage and propagation to other nodes, and enclave decryption), corresponding to *Steps 4*, *5*, and *11* in *Figure 16.7*, can be visualized in the following diagram:

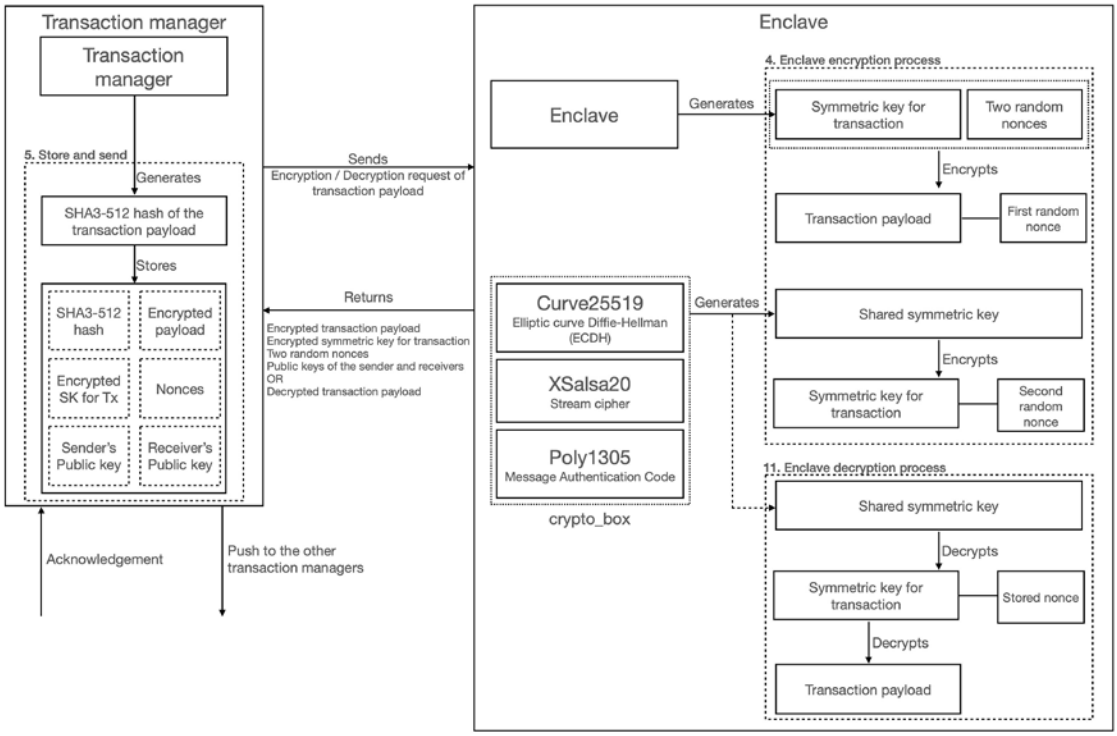



*Figure 16.8: Enclave and transaction manager high-level architecture*

We've covered quite a lot of theory here, but with this, we have now completed our introduction to Quorum private transactions. Next, we'll explore how access control works in Quorum.

## Access control with permissioning

As an enterprise blockchain platform, **Quorum** comes with an enterprise-grade access control mechanism that manages network membership for consortium members. It is implemented in smart contracts written in the **Solidity** language. It supports many features that a typical enterprise-grade permissioning mechanism would. It includes features to support the management of nodes, account-level permissioning, and support for voting-driven decision-making for permissioning actions.

The overall architecture of Quorum's permissioning mechanism can be visualized using the following diagram:

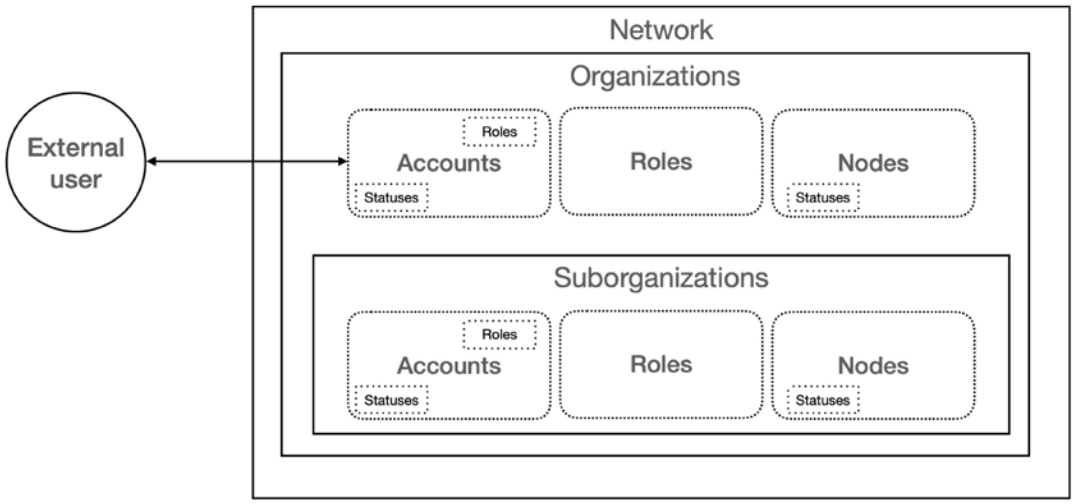



*Figure 16.9: Quorum permissioning mechanism*

This permissioning mechanism is inspired by a widely accepted industry standard called **Role-Based Access Control** (**RBAC**). It is standardized by the **American National Standards Institute** (**ANSI**) and is used in most enterprise-grade software, such as operating systems and cloud systems.

The RBAC standard is available here:

`https://standards.incits.org/apps/group_public/project/details.php?project_id=1658`.

The features that Quorum's permission mechanism supports are listed here:

- Role management
- Assignment of roles to accounts (subjects)
- Permissions management
- Node management

A role can be defined as a named job function in an organization.

At a fundamental level, the permissions management model supports granular control over:

- Funds transfer
- Smart contract creation
- Smart contract execution

It controls which account can transfer funds, create smart contracts, or execute smart contracts. These permissions are used to create roles, which are then assigned to accounts to enforce permissions.

Permissions are divided into two broad categories: *account permissions*, which control what functions an account can perform, and *node permissions*, which control the membership of nodes on the network.

Accounts can be assigned with different roles and statuses, whereas nodes can be assigned with statuses. Statuses for accounts include several statuses such as active, inactive, suspended, and blacklisted. Nodes can be in approved, deactivated, pending approval, or blacklisted states. The access types can be read-only, transact, contract deploy, and full access.

Now that we have covered how enterprise-grade permissioning is implemented in Quorum, let's now move on to another important aspect of enterprise blockchain: performance.

## Performance

Performance is a vast subject and can mean different things in different scenarios. Mostly, it is merely a measure of how many transactions a system can perform in a given interval, usually measured in **transactions per second** (**TPS**). It can also mean the overall throughput of the system, the quality of the service, and the ability to scale. From a business perspective, it could mean business performance. However, here, we will only deal with the TPS scenario and present some studies that have been performed to evaluate the performance of the Quorum blockchain.

The core idea behind performance enhancement in the Quorum blockchain is the usage of deterministic and fast consensus algorithms. Quorum supports various consensus mechanisms suitable for consortium networks. As compared to other public blockchain consensus mechanisms, the algorithms used in Quorum provide better performance.

Several evaluations have been made and reported regarding Quorum's performance. These studies have reported Quorum's performance to be as high as 2,500 TPS.

These studies are available here:

`https://scandiweb.com/blog/jpmorgan-s-quorum-blockchain-performance-testing`

`https://arxiv.org/pdf/1809.03421.pdf`

Now, we'll introduce the consensus mechanisms supported by Quorum.

## Pluggable consensus

Quorum supports several consensus algorithms that can be plugged in, based on use case requirements. For example, if the requirement is simple crash-tolerance, users can choose Raft. If more security is required and Byzantine faults need to be handled, then users can choose IBFT. We have listed the consensus mechanisms available in Quorum here:

- **Raft**: A crash fault-tolerant consensus algorithm
- **IBFT**: A PBFT-inspired BFT algorithm
- **Clique:** PoA, inherited from public Ethereum

We discussed all these algorithms in detail in *Chapter 5*, *Consensus Algorithms*. You can review these topics in detail there.

Now that we have gone through several features and have built a theoretical understanding of various features of the Quorum enterprise blockchain, let's now see how we can set up a Quorum network.

# Setting up a Quorum network with IBFT

In this section, we will set up a **Quorum** network with four nodes, demonstrating how an IBFT network can be set up and carry out private transactions in Quorum. These steps can be performed manually, or we can use a tool called **Quorum Wizard** to spin up a Quorum network quickly and easily.

For details on the manual setup, you can refer to the detailed Quorum setup instruction on Quorum's official website here: `https://docs.goquorum.consensys.net/tutorials/quorum-dev-quickstart`. We will use Quorum Wizard to set up our network.

## Installing and running Quorum Wizard

Quorum Wizard is a command-line tool written in **JavaScript** that enables users to create a local network of Quorum nodes quickly. It runs as an `npm` module and, as such, requires Node.js and npm to run.

Node.js can be installed from this address: `https://nodejs.org/en/`.

To check the current version installed, issue the following command:

```
$ npm -v
```

This command will produce the following output, indicating the installed version of the node package manager, `npm`:

```
6.14.4
```

This command will show the currently installed version of `node`:

```
$ node -v
```

This will produce an output specifying the version of `node` you have installed:

```
v12.18.0
```

If the expected version number is displayed after executing both of these commands, then we are all set to install Quorum Wizard.

Quorum Wizard can be installed using `npm install`. It's an excellent tool and cuts installation time down from hours to minutes:

```
$ npm install -g quorum-wizard
```

This will show an output similar to the one shown here:

```
/usr/local/bin/quorum-wizard -> /usr/local/lib/node_modules/quorum-wizard/
build/index.js
+ quorum-wizard@1.1.0
added 155 packages from 143 contributors in 27.897s
```

Once installed, we can run it with the following procedure.

We will create a 4-node IBFT network in this example. Simply run Quorum Wizard and follow the steps as prompted:

```
$ quorum-wizard
```

Running Quorum Wizard will show the following text:

```
Welcome to Quorum Wizard!

This tool allows you to easily create bash, docker, and kubernetes files to start up a quorum network.
You can control consensus, privacy, network details and more for a customized setup.
Additionally you can choose to deploy our chain explorer, Cakeshop, to easily view and monitor your network.

We have 3 options to help you start exploring Quorum:

  1.  Quickstart - our 1 click option to create a 3 node raft network with tessera and cakeshop

  2.  Simple Network - using pregenerated keys from quorum 7nodes example,
      this option allows you to choose the number of nodes (7 max), consensus mechanism, transaction manager, and the option to deploy cakeshop

  3.  Custom Network - In addition to the options available in #2, this selection allows for further customization of your network.
      Choose to generate keys, customize ports for both bash and docker, or change the network id

Quorum Wizard will generate your startup files and everything required to bring up your network.
All you need to do is go to the specified location and run ./start.sh

  (Use arrow Keys)
> Quickstart (3-node raft network with tessera and cakeshop)
  Simple Network
  Custom Network
  Exit
```

*Figure 16.10: Quorum Wizard options*

Select **Simple Network** and select all the options by using the up and down arrow keys. Press *Enter* after a choice is made, which will move us on to the next question in Wizard:

```
Simple Network
? Would you like to generate bash scripts, a docker-compose file, or a
kubernetes config to bring up your network? bash
? Select your consensus mode - istanbul is a pbft inspired algorithm with
transaction finality while raft provides faster blocktimes, transaction
finality and on-demand block creation istanbul
? Input the number of nodes (2-7) you would like in your network - a minimum of
4 is recommended 4
? Which version of Quorum would you like to use? Quorum 2.6.0
```

```
? Choose a version of tessera if you would like to use private transactions in
your network, otherwise choose "none" Tessera 0.10.2
? Do you want to run Cakeshop (our chain explorer) with your network? Yes
? What would you like to call this network? 4-nodes-istanbul-tessera-bash
```

Once all the options have been selected, as shown in the preceding code snippet, the tool will download and install dependencies. This process will produce an output similar to the one shown here. For brevity, the full output is not shown:

```
Downloading dependencies...
.
.
Building network directory...
Generating network resources locally...
Building qdata directory...
Writing start script...
Initializing quorum...
Done
```

Finally, the output shown here is produced, indicating the success of the operation:

```
Quorum network created
Run the following commands to start your network:
cd network/4-nodes-istanbul-tessera-bash
./start.sh
A sample simpleStorage contract is provided to deploy to your network
To use run ./runscript.sh public-contract.js from the network folder
A private simpleStorage contract was created with privateFor set to use Node
2's public key: QfeDAys9MPDs2XHExtc84jKGHxZg/aj52DTh0vtA3Xc=
To use run ./runscript private-contract.js from the network folder
```

Now, change to the `4-nodes-instabul-tessera-bash` directory, as shown here, and start the network:

```
$ cd network/4-nodes-istanbul-tessera-bash
$./start.sh
```

This will produce an output similar to the one shown here. Some of the output has been truncated for brevity:

```
Starting Quorum network...
.
.
.
All Tessera nodes started
Starting Quorum nodes
Starting Cakeshop
```

```
Cakeshop started at http://localhost:8999
Successfully started Quorum network.
```

And that's it! We now have a Quorum network with four nodes running on the IBFT protocol.

Now, let's do an experiment to see how private transactions can be created and run on the Quorum network. Quorum Wizard has already created the relevant scripts.

## Running a private transaction

In this section, we will explore how private transactions can be created and executed on a Quorum network:

1. From within the `4-nodes-instabul-tessera-bash` directory, run the command shown here:

```
$ 4-nodes-istanbul-tessera-bash ./runscript.sh private_contract.js
```

2. Once the preceding command runs, it will produce an output similar to the one shown here, indicating that the transaction has been sent and that a transaction hash `0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4` has been produced:

```
Contract transaction send: TransactionHash:
0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4
waiting to be mined...
True
```

Here, we have basically executed a private transaction and deployed a private smart contract that is only visible to Node 1 and Node 2. Nodes 3 and 4 are not privy to this transaction and they shouldn't be able to view the contents of the code or the values (state) of the contract. We will demonstrate in this example that this is indeed the case.

## Attaching Geth to nodes

Next, we'll attach Geth to each node, as shown using the commands in the following sections, and run the console commands as shown in the examples for each node.

**Node 1**: In order to interact with `geth` via IPC, enter the following command in the operating system's Terminal. This will open the Geth JavaScript console, where users can interact with the blockchain using several methods exposed by Geth:

```
$ geth attach qdata/dd1/geth.ipc
```

Once the `geth` console is open, enter the following command, which declares a variable named `abi`:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
```

```
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256"}],
"payable":false,"type":"constructor"}];
```

Now, create a new contract instance using the command shown here. We can use this object to access all methods and events of the contract:

```
> var simpleContract = eth.contract(abi)
```

Next, we use the instance object created previously to get a complete abstraction of our contract by using the command shown here:

```
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

Now, we can use the `simple` object to access all the methods in our smart contract. In the following example, we're using the `get()` method to return the stored value in our contract:

```
> simple.get()
42
```

This command outputs `42`, which is our stored value.

Note that after each statement we enter in the `geth` console, except the last statement, `simple.get()`, we also see that a message of `Undefined` is displayed. Simply ignore this; it is just a standard way in JavaScript of indicating uninitialized variables, non-existent object properties, or other similar scenarios.

**Node 2:** Similar to Node 1, open the `geth` console for Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

As we did for Node 1, in the `geth` console, enter the following commands:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256
"}],"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
42
```

Much like Node 1, the final command outputs `42`, which is our stored value.

**Node 3:** Again, similar to Node 1, open the `geth` console for Node 3 using the command shown here:

```
$ geth attach qdata/dd3/geth.ipc
```

In the geth console, enter the following:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256 "}],
"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
0
```

Note that the value returned is `0`, which indicates that Node 3 cannot access the value of `42`, which is expected as Node 3 is not privy to the transaction.

**Node 4:** Finally, we open the geth console for Node 4 using the command shown here:

```
$ geth attach qdata/dd4/geth.ipc
```

In the console, enter the following command as we did for nodes 1, 2, and 3:

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData",
"outputs":[{"name":"","type":"uint256"}],"payable":false,
"type":"function"},{"constant":false,"inputs":[{"name":"x","type":"uint256"}],
"name":"set","outputs":[],"payable":false,"type":"function"},{"constant":true,
"inputs":[],"name":"get","outputs":[{"name":"retVal","type":"uint256"}],
"payable":false,"type":"function"},{"inputs":[{"name":"initVal",
"type":"uint256 "}],"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
0
```

As expected, Nodes 3 and 4, which are not privy to the transaction, will see `0` as the transaction value. On the other hand, Nodes 1 and 2 are able to see the transaction value, which is `42`.

Remember, as we discussed earlier in this chapter, we need to monitor and visualize enterprise blockchain networks. Cakeshop fulfills that need.

## Viewing the transaction in Cakeshop

Cakeshop is a powerful visualization and administration tool with different features such as node management, block explorer, and contract management. Cakeshop is installed as part of the Quorum Wizard network creation process. Once the network runs successfully, we can browse to `http://localhost:8999`, where Cakeshop runs.

Cakeshop is shown in the following screenshot:

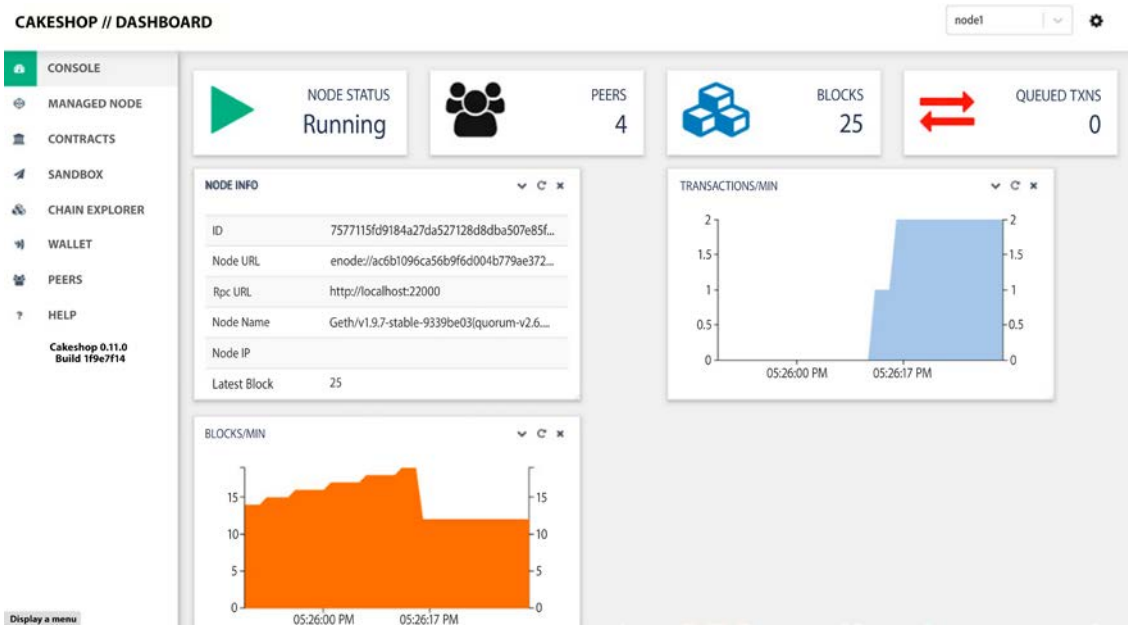



*Figure 16.11: Cakeshop*

In Cakeshop, we can see the transaction and relevant attributes. We simply browse to chain explorer and enter the transaction hash `"0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4"` from the *Running a private transaction* section. Cakeshop will display the status of the transaction, block ID, block number, contract address, and several other attributes, which makes it easy to explore the transaction process in detail:

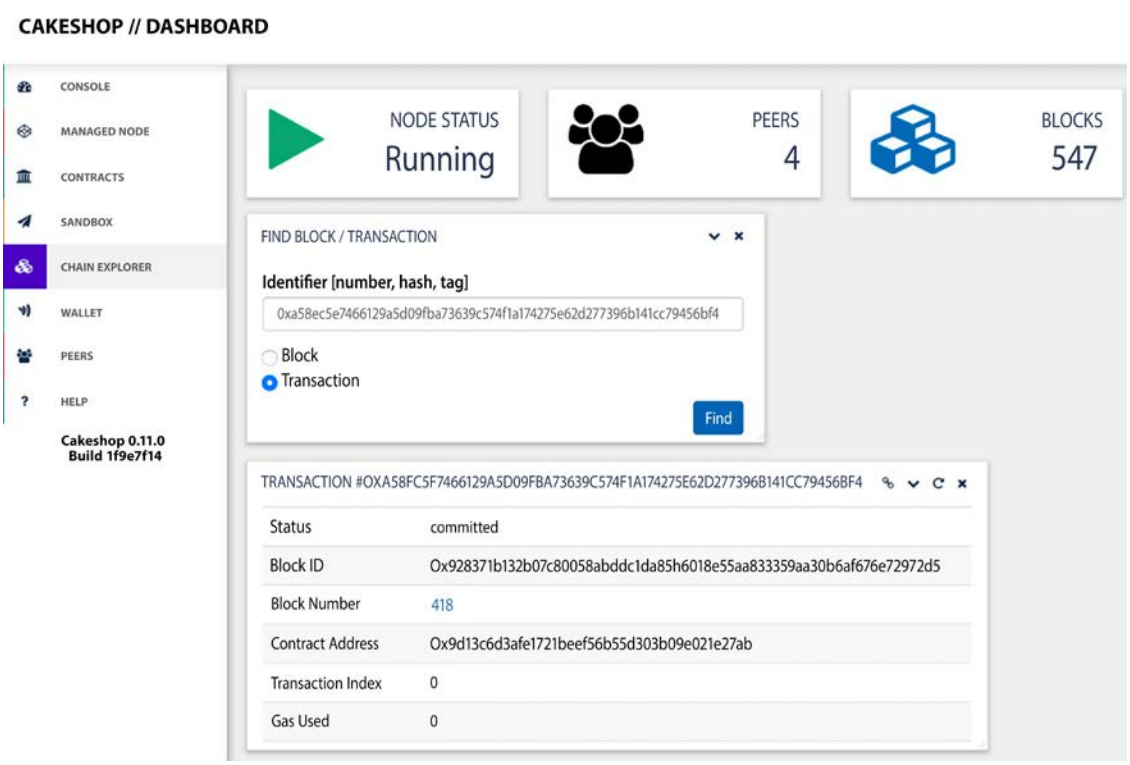



*Figure 16.12: Cakeshop chain explorer*

## Further investigation with Geth

We can also see the transaction receipts and contract code using the `geth` console.

For this, we `attach` to `geth` using the following command:

```
$ geth attach
```

We will do this on each node, as follows.

**Node 1**: First, we open the `geth` console using the following command on Node 1:

```
$ geth attach qdata/dd1/geth.ipc
```

When the `geth` console opens, enter the following statement, which uses the transaction hash from our first step of contract deployment:

```
> eth.getTransaction("0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b14
1cc79456bf4")
```

The output shown here will be displayed:

```
{
  blockHash:
"0x92837fb132b07c80058abddc1da85b6018e55aa833359aa30b6af676e72972d5",
  blockNumber: 418,
  from: "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
  gas: 4700000,
  gasPrice: 0,
  hash: "0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4",
  input: "0x6d99da7776195f534b48d53e8a2e94231299d38967a874115fe329b0a8a236e83258
b04187674022ab7bb79847e45d02d266009ccb37a1011967e77da12b152b",
  nonce: 2,
  r: "0x3ca297d21fed45e3701d2e3ed9d4e3a144c1350ea5b981005ff734efc8467967",
  s: "0x6233d5d65445ae258e7c04efd085f1cdba42ca5f63c9982b4db779d45d098bae",
  to: null,
  transactionIndex: 0,
  v: "0x26",
  value: 0
}
```

In the preceding output, we can see that the `V` value is `0x26` in hexadecimal, which means `38` in decimal. This value of `38` indicates that the transaction is private.

Also, notice the input field, which is the encrypted payload of the transaction.

Similarly, we can get the transaction receipt of this transaction by issuing the call shown here:

```
> eth.getTransactionReceipt("0xa58ec5e7466129a5d09fba73639c574f1a174275e62
d277396b141cc79456bf4")
{
  blockHash:
"0x92837fb132b07c80058abddc1da85b6018e55aa833359aa30b6af676e72972d5",
  blockNumber: 418,
  contractAddress: "0x9d13c6d3afe1721beef56b55d303b09e021e27ab",
  cumulativeGasUsed: 0,
  from: "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
  gasUsed: 0,
  logs: [],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000",
  status: "0x1",
  to: null,
  transactionHash:
"0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4",
  transactionIndex: 0
}
```

We can see in the output that the status is `0x1`, indicating the success of the transaction. As we need to get the code of this transaction (smart contract), we need the contract address, which we can see in the `contractAddress` field. We copy that address into the RPC call shown here to fetch the code for this contract:

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
"0x60606040526000357c0100000000000000000000000000000000000000000000000000000000
00000900463ffffffff1680632a1afcd914605157806360fe47b11460775780636d4ce63c1
46097575b600080fd5b3415605b57600080fd5b606160bd565b604051808281526020019150
5060405180910390f35b3415608157600080fd5b6095600480803590602001909190505060
0c3565b005b341560a157600080fd5b60a760ce565b604051808281526020019150506040
180910390f35b60005481565b806000819055505b50565b6000805490505b905600a165627
a7a72305820d5851baab720bba574474de3d09d beaabc674a15f4dd93b974908476542c23
f00029"
```

As we can see in the preceding output, the contract code is visible as bytecode in hex.

**Node 2**, which is privy to the transaction: Similarly, to Node 1, open the `geth` console on Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

In the `geth` console, enter the following statement. This method takes the address of the smart contract and returns the compiled smart contract code (bytecode):

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

The following output shows the bytecode of our contract at address `"0x9d13c6d3afe1721beef56b55d303b09e021e27ab"`:

```
"0x60606040526000357c0100000000000000000000000000000000000000000000000000000000
00000900463ffffffff1680632a1afcd914605157806360fe47b11460775780636d4ce63c1
46097575b600080fd5b3415605b57600080fd5b606160bd565b604051808281526020019150
5060405180910390f35b3415608157600080fd5b6095600480803590602001909190505060
0c3565b005b341560a157600080fd5b60a760ce565b604051808281526020019150506040
180910390f35b60005481565b806000819055505b50565b6000805490505b905600a165627
a7a72305820d5851baab720bba574474de3d09d beaabc674a15f4dd93b974908476542c23
f00029"
```

As expected, the code is also visible on Node 2, as Node 2 is a party to the transaction.

**Node 3**, which is not privy to the transaction: Now, we open the `geth` console on Node 3:

```
$ geth attach qdata/dd3/geth.ipc
```

In the console, enter the following statement with the address of our smart contract:

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

This will show the output shown here:

```
"0x"
```

This output indicates that the contract code is only available on the nodes that are party to the transaction. The contract code is not available on other nodes that are not a party to the transaction.

This experiment demonstrates that the contract is only available on the nodes that are party to the transactions. First, we deployed our private contract from Node 2 with Node 1 as a participant and used the `geth` console to interact with the contract. We saw that the value `42` is only available on the nodes that are party to the transaction. Also, we further experimented and noticed that not only the value but also the smart contract code is only available on the nodes that are party to our private contract (transaction).

This network is now available for further experiments. You can create your own private contracts with different nodes as parties. You can also explore how other methods such as `debug_traceTransaction` behave when interacting with private contracts. This network can also be used to build some PoCs for an enterprise dApp.

After this quick experiment of demonstrating how private transactions work in Quorum, let's explore other Quorum projects.

# Other Quorum projects

As mentioned earlier, Quorum is a feature-rich platform and is under continuous improvement and development. As a result, new features and projects are introduced regularly. Some of the other projects under Quorum are listed here, along with brief descriptions.

## Remix plugin

This is a plugin for the popular Remix IDE, which supports private smart contracts on the Quorum blockchain. More information on this is available here: `https://docs.goquorum.consensys.net/tutorials/quorum-dev-quickstart/remix#docusaurus_skipToContent_fallback`.

## Pluggable architecture

Quorum supports a pluggable architecture that allows us to add new features to the Quorum client as plugins. This approach allows us to keep the core Quorum services separate from the new features, which allows greater modularity and extensibility without modifying the core client. More information on this feature is available here: `https://docs.goquorum.consensys.net/concepts/plugins`.

Quorum is a large project with many excellent enterprise features. It is not possible to cover all of them in this chapter. However, the material provided should get you started with the Quorum blockchain setup, plus an understanding of the concept of private transactions.

There is a wealth of information available in the official Quorum documentation at `https://docs.goquorum.consensys.net/en/latest/`. You are encouraged to go through the documentation to get a deeper understanding of Quorum.

Quorum has been used in many industries for different projects, including finance, supply chain, healthcare, media, and government sectors. A couple of example uses are:

- Tracking luxury goods: `https://www.coindesk.com/louis-vuitton-owner-lvmh-is-launching-a-blockchain-to-track-luxury-goods`
- Post-trade processing platform: `https://cointelegraph.com/news/oil-trading-blockchain-platform-vakt-launches-with-shell-bp-as-first-users`

Quorum is also available on different cloud platforms, including:

- Azure: `https://azure.microsoft.com/en-us/solutions/web3/#overview`
- Kaleido: `https://www.kaleido.io`

# Summary

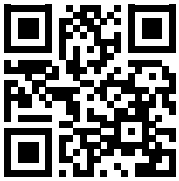
In this chapter, we started with an introduction to enterprise solutions and blockchain. We saw some limiting factors in public chains that make them unsuitable for enterprise use cases. We also looked at some requirements that, when met, will make a blockchain suitable for enterprise use cases.

We also covered how to design enterprise blockchain solutions and made a case to see the enterprise blockchain solutions in the context of the enterprise architecture. Next, we explored cloud computing and the definition of **Blockchain as a Service**. We briefly looked at the efforts being made to standardize enterprise Ethereum specifications. The last sections of this chapter covered enterprise blockchain platforms, including Quorum, and finally, we set up an IBFT network using Quorum to demonstrate how privacy is achieved using Quorum. Recent blockchains, such as VMware Blockchain, were also introduced.

In the next chapter, we will introduce scalability, security, and other challenges that blockchains face, how many of these limitations are being addressed, and what the future holds for blockchain technology.

## Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`